Defensive JavaScript

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Sergio Maffeis

Defensive JavaScript

Building and Verifying Secure Web Components

Karthikeyan Bhargavan[1], Antoine Delignat-Lavaud[1], and Sergio Maffeis[2]

[1] INRIA Paris-Rocquencourt
[2] Imperial College London

Abstract. Defensive JavaScript (DJS) is a typed subset of JavaScript that guarantees that the functional behavior of a program cannot be tampered with even if it is loaded by and executed within a malicious environment under the control of the attacker. As such, DJS is ideal for writing JavaScript security components, such as bookmarklets, single sign-on widgets, and cryptographic libraries, that may be loaded within untrusted web pages alongside unknown scripts from arbitrary third parties. We present a tutorial of the DJS language along with motivations for its design. We show how to program security components in DJS, how to verify their defensiveness using the DJS typechecker, and how to analyze their security properties automatically using ProVerif.

1 Introduction

Since the advent of asynchronous web applications, popularly called AJAX or Web 2.0, JavaScript has become the predominant programming language for client-side web applications. JavaScript programs are widely deployed as scripts in web pages, but also as small storable snippets called bookmarklets, as downloadable web apps,[3] and as plugins or extensions to popular browsers.[4] Mainstream browsers compete with each other in providing convenient APIs and fast JavaScript execution engines. More recently, Javascript is being used to program smartphone and desktop applications[5], and also cloud-based server applications,[6] so that now programmers can use the same idioms and libraries to write and deploy a variety of client- and server-side programs.

As more and more sensitive user data passes though JavaScript applications, its confidentiality and integrity becomes and important security goal. Consequently, JavaScript applications rely on a number of security libraries for cryptography and access control. However, neither the JavaScript language nor its execution environment (e.g. the web browser) are particularly well suited for security programming. For example, to aid uniform deployment across different

[3] `https://chrome.google.com/webstore/category/apps`
[4] `https://addons.mozilla.org/`
[5] `http://dev.windowsphone.com/develop`
[6] `http://node.js`

browsers, JavaScript allows a number of core language primitives to be redefined and customized. This means that a JavaScript security library that may run alongside other partially-trusted libraries must take extra care so that its functionality is not subverted and its secrets are not leaked.

In this tutorial, we investigate approaches to build and verify JavaScript programs that implement security-criticial tasks, such as cryptographic protocols. Our programs must contend not just with the traditional network attacker, but also with a variety of web-specific attacks, such as malicious hosting websites and Cross-Site Scripting (XSS). In other words, not just the communication channel but even parts of the execution environment may be under the control of the adversary. We propose a typed subset of JavaScript, called Defensive JavaScript, that enables formal security guarantees for programs even in this threat model. Our language and verification results previously appeared in [12].

Many existing works investigate the security of formal models of web application protocols [3, 17, 8], but none of them can provide concrete security guarantees for JavaScript code. Still, we build upon these prior results (especially [8]) to develop our threat model and verification techniques. Another closely related line of work investigates the use of type-preserving compilers to generate JavaScript programs that are secure-by-construction [18, 25]. We will focus only on language-based protections in JavaScript, but note that HTML-level isolation techniques may also be effectively used to separate trusted web security components from untrusted JavaScript [4].

In the rest of this section, we will seek to better understand the threat model and security goals of JavaScript security components through three examples.

1.1 Encrypted Cloud Storage Websites

Storage services (e.g. Dropbox) allow users to store their personal files on servers hosted within some cloud infrastructure. Since users often rely on these services to back up important files and share them across devices, the integrity and confidentiality of this data is an important security requirement, especially since the cloud servers may be under the control of a third party. Consequently, mainstream storage services typically encrypt user files before storing them in the cloud. A hacker who breaks into the cloud server to obtain the encrypted files would also need to steal the file encryption key from the storage service.

Some services, such as SpiderOak and MEGA, seek to provide a stronger privacy guarantee to their users, sometimes called *host-proof* hosting–even if the storage service and its cloud servers are both hacked, the user's files should remain confidential. The key mechanism to achieve this goal is that a user's file encryption keys are generated and stored on the client-side; even the storage service does not get to see it, and so cannot accidentally leak it.

For example, to access their files stored on MEGA, users visit the MEGA website, which downloads and runs a JavaScript program in the browser. The program asks the user for a master passphrase, derives an authentication token and an encryption key from the passphrase, and sends the username and authentication token to the website. If the token matches the username, the web

page allows the user to download or upload encrypted files from a cloud server. The JavaScript program encrypts and decrypts user files upon request, using the encryption key derived from the master passphrase, but the key and the passphrase never leave the browser.

Hence, the storage service implements an application-level cryptographic protocol in JavaScript. This programming pattern is also popular with other security web applications such as password managers (more below) and with privacy-preserving websites like ConfiChair [5] and Helios [1].

The main threat to this design is that if the attacker manages to inject a malicious script into the website, that script will be able to steal the master passphrase (and hence the user's files). This script injection may be achieved by hacking the web server, or by tampering with externally loaded scripts on the network, or by exploiting a cross-site scripting (XSS) attack. Many such attacks have been found in previous work [10, 7, 12] and reports from the MEGA bug bounty program indicate that such attacks are a common concern. Cloud storage websites employ many techniques to block these attack vectors, such as Strict Transport Layer Security [20] and Content Security Policy [24], but the increasing incidence of server-side compromises, man-in-the-middle attacks on TLS, and XSS vulnerabilities on websites, indicates that it would be prudent to try to protect user data even if the website had a malicious script.

Even ignoring malicious scripts, to provide any formal security guarantee for a website security component that runs alongside unknown third party scripts, the component would need to be robust against bugs in these scripts. To give a concrete example, the MEGA website relies on about 70 scripts, and less than 10% of their code is related to cryptography; most of the rest implements the user interface. So the correctness of the cryptographic library and the secrecy of its keys relies on the good behavior of these UI scripts, which are not written by security experts and may be difficult to formally review.

1.2 Password Manager Bookmarklets and Browser Extensions

Password managers (e.g. LastPass) help users manage and remember their passwords (and other sensitive data such as credit card numbers) for use at various websites. They are often implemented in JavaScript and deployed as a browser extension or bookmarklet that detects the login page of a website, looks up a password database for a matching username and password, and offers to fill it in automatically. If there is no matching password, it may offer to generate a difficult-to-guess password and store it in the database. To synchronize and backup the password database across a user's devices, many password managers implement the host-proof encrypted cloud storage pattern described above.

For example, LastPass users can generate a "Login" bookmarklet and add it to their browser's bookmarks. The bookmarklet contains a JavaScript program embedded with an encryption key for the user's password database. When a user next visits the login page at some website, she may click on the bookmarklet to automatically log in. Clicking on the bookmarklet executes its JavaScript program in the scope of the current page. The program contacts the LastPass

website and retrieves the currently logged-in LastPass user's encrypted password data from the cloud server. It then uses the encryption key embedded in the bookmarklet code to decrypt the password for the current page and fills in the login form. If the browser does not have an active login session with LastPass, the bookmarklet has no effect.

The main threat to the bookmarklet design is that it may be clicked on a malicious website that may then tamper with the JavaScript environment to subvert the bookmarklet's functionality. A typical case is if the user accidentally clicks the bookmarklet on a website that looks like a known trusted site. Or the user has passwords for two differen sites stored in her database, and one of them may have been compromised. In these situations, the main goal of the malicious website is to steal password of the user at a different honest website. The bookmarklet tries to prevent such attacks by identifying the website the bookmarklet has been clicked on and only using its embedded secret on trusted websites. However, identifying the host website and protecting the bookmarklet secret are difficult in a tampered JavaScript environment, leading to many attacks [2, 10, 7, 12]. We propose a programming discipline that enables secret-keeping bookmarklets that are robust against tampered environments.

As an alternative to bookmarklets, many password manage also provide a downloadable browser extension that executes a similar JavaScript program, but in a safer, more isolated JavaScript context. Password manager browser extensions are subject to their own threat model [9], not detailed here. In particular, even extensions must protect their secrets from being leaked by bugs in other included JavaScript programs. To give a concrete example, the LastPass extension for the Google Chrome browser has 119 JavaScript files, of which only 5 contain any cryptography, but their security guarantees still must rely on the correctness of these other scripts.

1.3 Single Sign-On and Social Sharing Buttons

Single Sign-On protocols (e.g Facebook's Login button) are widely used by websites that wish to implement authenticated sessions without the hassle of user registration and password management. Another advantage is that the website can leverage their users social networks to provide a richer experience (e.g. Facebook's Like button). From the user's viewpoint, single sign-on and social sharing buttons offer her a convenient and secure way of accessing and sharing data across different websites, without needing to remember different passwords.

For example, to include the Facebook Login button on a web page, a website W loads a JavaScript library provided by Facebook that displays the button. When a user clicks on the button, the program asks Facebook for the currently logged-in user's access token for the current website W. If the user is logged in and has previously authorized Facebook to provide an access token to W, Facebook returns the access token in a URL. Otherwise, the user is forwarded to a page where she may login and authorize W (or not.) The program then gives the access token to the website and also provides functions to access the Facebook API and read or write (authorized elements of) the current user's social profile.

The protocol implemented by Facebook is OAuth 2.0 [19], which also prescribes other message flows for server-side tokens and smartphones. Other popular single sign-on protocols, such as OpenID, SAML, and BrowserID, provide similar message flows that websites may use to obtain access tokens as user-specific credentials.

The main threat to the single sign-on interaction above is that the access token may be stolen by a malicious website and then used to impersonate the user at an honest website, or to read or write the user's profile information on her social network. The OAuth 2.0 flow is particularly vulnerable since access tokens are sent in URLs which may be leaked by Referrer headers, or by HTTP redirection, or by various browser and application bugs [8, 26, 12]. Since the access token is used as a bearer token, and is often not specific to a website, it can be immediately used by the adversary on any website to impersonate the user.

The BrowserID single sign-on protocol seeks to mitigate the effects of token theft by using public key cryptography to authenticate the client.[7]. Mozilla's implementation of BrowserID is written fully in JavaScript. The client includes a JavaScript cryptography library that may be included by any site to retrieve and sign tokens on behalf of the user. Even the single sign-on server is written in JavaScript and deployed over `node.js`. The design of BrowserID has been carefully evaluated by formal analysis [17], but to prove the code correct, one must show that all the scripts loaded alongise behave safely. In Mozilla's implementation, the server-side protocol moduls is loaded among 158 other `node.js` modules, and a bug or malicious function in any of these modules could compromise both ther server's and user's private keys.

1.4 Towards Verifiably Secure Web Components

We have discussed three popular categories of JavaScript security components that seek to protect sensitive user data such as files, passwords, and access tokens from malicious websites using various combinations of authentication protocols and cryptography. Each of these components is used in conjunction with a number of other scripts that may modify the JavaScript environment.

Our goal is to write JavaScript security components in a style that their security can be formally proved even if the context is malicious. In particular, we aim for a language-level isolation guarantee for our programs–that their input-output functional behavior cannot be tampered with by the environment. As a corollary, any secrets that are correctly protected by cryptography in our programs cannot be stolen or modified by the adversary. This simple-sounding isolation guarantee would be trivial to obtain in traditional programming langugages with sound type systems, such as OCaml and Java. However, the flexibility of JavaScript breaks many guarantees presumed by the programmer and the language must be reined in before we can achieve our goal.

In Section 2, we discuss the peculiarities of JavaScript and the browser environment that make it difficult to isolate security components. In Section 3, we

[7] `http://login.persona.org`

present Defensive JavaScript (DJS), a typed subset of JavaScript that guarantees isolation from the environment. In Section 4, we present a large cryptographic library written in DJS and use it to write and verify simple cryptographic web applications. Section 5 concludes.

2 Secure Messaging in an Untrusted Environment

As a motivating example, we consider how to implement a JavaScript program that sends an authenticated message to a server. Our target web page is hosted on a website W at URL `http://W.com` and it loads three scripts:

```
<html>
 <body>
 <script src="attacker1.js"></script>
 <script src="messaging.js"></script>
 <script src="attacker2.js"></script>
 </body>
</html>
```

The first and third scripts are arbitrary malicious scripts chosen by the attacker. The second script is our program that provides an API to send messages to a server S at the URL `http://S.com`, via the `XMLHttpRequest` asynchronous messaging API provided by the browser. (In some cases, W may be the same site as S.) We assume that the program and S share a secret MACing key k. The program uses this key to attach a MAC to each message sent to S.

The security goal is message authentication: every message received and verified by S must have been sent by our program running at W. In particular, it should not be possible for the attacker scripts to steal the MAC key k and forge messages to S. The above web page scenario may seem too paranoid, but more generally, we want to guarnatee that that even if the surrounding scripts are just buggy, not malicious, they still cannot accidentally leak the key.

2.1 Secure Delivery of the Secret Key

The first challenge is to deliver the MAC key to `messaging.js` in a way that cannot be read by the other two attacker scripts.

Injecting the key as a token into the HTML document, or an HTTP cookie, or in browser local storage would not work; if the messaging script can read it, so can the attacker's script. The only safe place for the key is to embed it into the messaging program. But even in this case, there are many pitfalls. Consider the following messaging script `messaging.js` with a key included on top:

```
var key = k;
var api = function(msg){ .../*send authenticated message*/}
```

Unfortunately, the attacker script `attacker2.js` can simply read the variable `key` from the environment and obtain the key. A better solution would be to protect the key within the function:

```
var api = function(msg){
  var key = k;
  .../*send authenticated message*/
}
```

Now the script `attacker2.js` can no longer read the local variable `key`. However, it can retrieve the source code of the function `api` as a string by calling `api.toSource()`. It can then extract the embedded key k from the string. To protect the source code of the function, we need to rewrite the function by wrapping it within an anonymous function closure:

```
var api = (function (){
            var _api = function(msg){
                        var key = k;
                        .../*send authenticated message*/}
            return function(msg){return _api(msg);}
          )();
```

Now, calling `api.toSource()` only reveals the code of the wrapper function, and the code of the real `_api` function (which embeds the key k) remains private.

There remains another way for the attacker scripts to obtain the source code of `_api`. If the script `messaging.js` is served from the current website's origin `http://W.com`, the source code of the whole script can be retrieved by either attacker script by making an `XMLHttpRequest` to the script's URL:

```
var xhr = new XMLHttpRequest();
xhr.open("GET","http://W.com/messaging.js",false);
xhr.send();
var program = xhr.responseText;
```

To prevent this, the messaging script must be served from a separate origin. For example, the website W could set aside a separate origin for serving only scripts, and place the messaging script at say `http://scripts.W.com/messaging.js`. in our example, it would also be suitable to source it from S's origin, say at `http://S.com/messaging.js`, so S can inject the shared key into the script. In both cases, the attacker scripts on `http://W.com` would be unable to make an `XMLHttpRequest` to read the code, due to the Same Origin Policy.

2.2 Calling External Functions

To construct and send a message, our messaging program will rely on several external functions either builtin to the JavaScript language or provided by the browser as part of the DOM library. For example, commonly used string functions such as concatenation (`s.concat(t)`) or search (`s.indexOf(t)`) are defined as methods in the `String` prototype. Other useful functions on arrays and objects are provided by the `Array` and `Object` prototypes. The `window.Math` object provides implementations of many mathematical functions. The `XMLHttpRequest` object allows asynchronous messaging with remote servers, and `postMessage` API implements client-side messaging between windows. Finally, the `document` object

(or DOM) provides functions for reading and writing the HTML document (e.g. `document.getElementById(''body'')`).

These external library functions are widely used by JavaScript programs. However, in our threat scenario, the attacker script `attacker1.js` may have redefined every one of these functions by modifying the `String`, `Array`, and `Object` prototypes, or by redefining these functions and objects in the `window` and `document` objects. For example, the following code redefines the `XMLHttpRequest` object, so that all messages send by the messaging script can be intercepted:

```
window.XMLHttpRequest =
  function(){
   return {open: function(){/*do whatever*/},
           send: function(){/*do whatever*/}}}
```

Suppose our messaging program is written as follows; in addition to the `XMLHttpRequest` object (and its methods), the code calls `Crypto.HMAC`:

```
var api = (function (){
            var _api = function(msg){
                        var key = k;
                        var xhr = new XMLHttpRequest();
                        xhr.open("GET","http://S.com",false);
                        xhr.send(Crypto.HMAC(key,msg) + "," + msg);
                       }
            return function(msg){return _api(msg);}}
          )();
```

This code exemplifies three dangers of calling an external function.

First, the call to `Crypto.HMAC` leaks the key, since the attacker may have redefined the function. Consequently, the only safe choice here is to inline the code of the `HMAC` function into the messaging program. The `HMAC` function in turn relies on a hashing function (say `SHA-256`) which would also need to be included within the program. (To see what these functions look like in JavaScript, see our implementation in Appendix A.)

Second, the call to any external function exposes `_api` function to a stack-walking attack. For example, the attacker can redefine `XMLHttpRequest.send` so that when it is called, it reads the source code of its calling function using the `caller` method in the `Function` prototype:

```
stackwalk = function(){var program = stackwalk.caller.toSource();...}
window.XMLHttpRequest =
  function(){
   return {open: stackwalk,
           send: stackwalk}}
```

Adding the above code in `attacker1.js` will set up the environment such that when `_api` calls `xhr.open`, the attacker obtains the source code of `_api` and hence its embedded key. The attack relies on the implementation of the `caller` method, and it it works at least in Firefox at the time of writing. More generally, this kind of stack-walking is a powerful attack vector. Whenever a function `f` is

called, it can access its caller by accessing `f.caller`, and the next level on the call stack by accessing `f.caller.caller`. At each level, it may examine (and even overwrite) the arguments of the function.

Third, if our messaging script ever calls an external function, the attacker may redefine its behavior so that the result of the function is not as expected. For example, `s.concat(t)` may always return a constant string or `Math.pow` may always return `0`. In such cases, the functional integrity of our script has been compromised, and if the results of these functions are used in the `MAC` function, the authentication protocol may be broken even without leaking the secret key.

In summary, any external function calls from a the messaging script may lead to a full compromise of its secrets and its functionality. To be safe, the script must never call functions from within security sensitive functions whose source code or arguments may be secret. Instead, all external function calls should be factored out into a top-level wrapper function that calls a self-contained API:

```
var api = (function (){
            var hmac = function(key,msg){/* inlined HMAC code */}
            var _api = function(msg){
                        var key = k;
                        return (hmac(key,msg) + "," + msg);
                       }
            return function(msg){return _api(msg);}
         )();
var msg_api = function (msg) {
               var mac = api(msg);
               var xhr = new XMLHttpRequest();
               xhr.open("GET","http://S.com",false);
               xhr.send(mac);
             }
```

Here, the external function call to `XMLHttpRequest` is performed outside the sensitive API by a function `msg_api` that has no access to the secret MACing key. Walking the stack to get to `msg_api` does not allow the attacker to steal any secrets or to tamper with the `_api` function.

2.3 Implicit Calls to External Functions

In addition to explicit function calls, many JavaScript constructs implicitly trigger methods defined in various prototypes. Since these prototypes may be modified by the adversary, we must also avoid such implicit calls in defensive code.

The first category of implicit function calls are *coercions*. For example, in the expression `e == e'`, if `e` is an object and `e'` is a number, then the equality will trigger an implicit coercion `e.valueOf` of `e` from string to number. This method `valueOf` is defined in the `String` prototype. More generally, comparison between any object and a string or a number may trigger the `valueOf` or `toString` methods in that object's prototype. Hence, by redefining these methods in the `Object` prototype, the attacker can intercept any function that triggers an implicit coercion and mount the attacks described in the previous subsection.

The second category of implicit function calls are *getters* and *setters*. Whenever an object is accessed at an undefined property (e.g. `o.x`), the JavaSript interpreter traverses the prototype hierarchy to see if the property `x` is defined in one of the prototypes that the object is derived from. If, say, none of the prototypes has defined `x`, but the `Object` prototype defines a getter function for `x`, then reading the property `o.x` will trigger this function. Similarly, if the `Object` prototype has a setter function for `x`, writing to `o.x` will call the setter.

By defining getters and setters for specific properties, an attacker script can cause trusted code to trigger an external function if it ever accesses an undefined property. Similarly, if an array or string is every indexed out of bounds, it may trigger a getter or setter in the `Array` prototype. Consequently, in our setting, the messaging program should never access arrays, strings, or objects outside their declared ranges. In particular, the popular JavaScript idiom of first declaring an empty object and then extending it is vulnerable to attack:

```
1 Object.defineProperty(Object.prototype,"a",{set:function(){...}});
2 var x = {};
3 x.a = 1; // triggers malicious setter
4 Object.defineProperty(Array.prototype,"0",{set:function(){...}});
5 var y = [];
6 y[0] = 1; // triggers malicious setter
7 Object.defineProperty(Array.prototype,"1",{get:function(){...}});
8 y[0] = y[1]; // should be undefined, but triggers malicious getter
```

A particular subcase of prototype poisoning is worth mentioning. JavaScript offers a `for...in` loop construct that goes through all the properties of an object. For example `for (i in {x:1})print(i)` is expected to print ``x'' and `for (i in [1])print(i)` is expected to print the single array index `0`. However, if the attacker modifies `Object` and `Array` prototypes to add more properties, those properties will also be printed here. Even checking that each property was defined locally within the object using the `Object.hasOwnProperty` function does not help, since this function could also be modified by the adversary.

2.4 Defensive Programming Idioms

We have discussed many potential attack vectors that a malicious script may employ when trying to subvert an honest JavaScript program running in the same environment. To prevent these attacks, we advocate a defensive programming discipline where programs aim to isolate their security-critical code from the environment by using function closures, by being loaded from a different origin, by refusing to explicitly call external functions, and by carefully preventing the triggering of coercions and prototype lookups. To systematically check our programs for all these isolation conditions, we propose a static type system. Defensiveness is a first step towards formal security guarantees. Once scripts like our messaging program are correctly isolated, we may rely on their context-independent semantics and on the functional integrity of their cryptographic libraries to build automated security verification tools.

Alternative Mitigations The injunction that the core messaging API must be fully self-contained may seem draconian and one may wonder if there are some cases in which calling external functions is safe. If the goal is only to prevent stack-walking, one may hide the stack by calling all external functions through a recursive wrapper function [25]. However, this requires a source-to-source translation to implement effectively, especially for object methods like `xhr.send`.

Recent versions of JavaScript give programs the ability to freeze objects and mark various properties as unmodifiable and/or unconfigurable (cannot be deleted). It is tempting to suggest that the website W should freeze some objects or that the browser should guarantee that some DOM properties are unforgeable. These objects and properties would then be safe to access. However, the problem with both `Object.freeze` and `Object.defineProperty` is that they need to apply to the top object in the object hierarchy, otherwise it is ineffective. For example, the properties `document.location.href` and `window.location.href` are commonly considered unforgeable since modifying them would take the webpage to a new location. Indeed, most browsers prevent JavaScript from redefining these properties. However, the attacker may directly redefine the `window.document` object (FireFox) or the `window.location` object (Internet Explorer).

Another option is for the website W to run a script first that makes copies of all relevant objects before they have been tampered by the attacker [18]. However, ensuring that a script runs first on a web page is surprisingly tricky [25]. Moreover, this solution does not work in scenarios where the website W itself may be malicious or compromised.

One may also use isolation mechanisms outside JavaScript, such as HTML `iframes` to effectively separate trusted and untrusted code [4]. In this paper, we do not investigate such mechanisms and instead focus only on language-based isolation. We note that the use of `iframes` relies on the semantics of the Same Origin Policy which remains to be fully standardized, let alone formalized [28]. Furthermore, `iframes` may not be available in some JavaScript runtime environments, such as smartphones and server applications. In these environments, defensive programming becomes necessary.

3 Defensive JavaScript

We present a subset of JavaScript that enforces a strong defensive programming discipline. Our language, Defensive JavaScript (DJS), imposes restrictions on JavaScript code both at the syntactic level and though a static type system. The main elements guiding the design of DJS are as follows:

Static Scopes The variable scoping rules of JavaScript are notoriously difficult to understand. For example, functions may use local variables before they are declared. More worryingly, if a JavaScript program ever accesses a variable that is not in its local scope, this access may trigger a getter or setter in some prototype object. Consequently, we require that all variables in DJS programs be strictly statically scoped. We impose this by restricting the